RV COLLEGE OF ENGINEERING

(Autonomous Institution affiliated to VTU, Belagavi)

Department of Computer Science and Engineering

Subjects: Operating System

Course Codes: CS235AI

Academic Year: 2023-2024

Report

Experiential Learning

Student Details

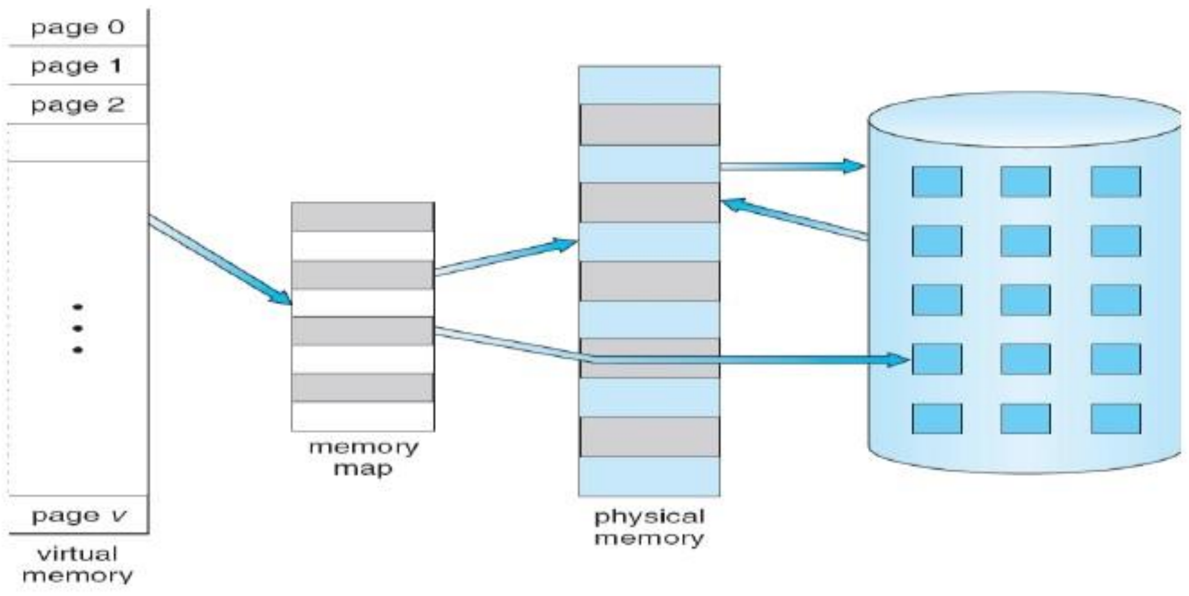
Title of Work	Implementing Virtual Memory Management System
Group Member Name List	VIKAS J (RVCE23BCS404) WILSON (RVCE23BCS418)
Branch	Computer Science and Engineering [CSE]
Section	CSE - D
Semester	III

Submitted to :
Dr. Jyoti Shetty
Assistant Professor

Contents

Problem Statement

Design and implement an efficient virtual memory management system for an operating system to address the challenges of optimizing page replacement policies, minimizing page faults, and enhancing overall system performance. The system should be capable of dynamically managing the allocation and deallocation of virtual memory space, ensuring seamless interaction between physical and virtual memory, and implementing effective page replacement strategies.



Introduction

In the realm of computer science, the concept of virtual memory stands as a cornerstone for modern operating systems (OS). Virtual memory is a powerful abstraction that allows programs to operate as if they have access to a vast and contiguous block of memory, despite the limitations of physical RAM (Random Access Memory). Virtual memory management plays a pivotal role in efficiently utilizing system resources and providing a seamless user experience. Operating Systems (OS) use virtual memory to abstract physical memory resources, enabling processes to access memory locations that may not necessarily reside in the main physical memory (RAM). This abstraction is crucial for multitasking environments where numerous processes compete for limited physical memory resources.

Here's a breakdown of the key components and functionalities of a virtual memory management system in an OS:

1. **Abstraction of Physical Memory**: Virtual memory provides a layer of abstraction over physical memory resources. Instead of directly addressing physical memory locations, processes interact with virtual memory addresses. This abstraction allows for efficient memory allocation and management, decoupling the logical view of memory from its physical implementation.

2. **Address Translation**: A fundamental aspect of virtual memory management is address translation. When a process references a virtual memory address, the OS translates this address into a corresponding physical address. This translation is facilitated by the Memory Management Unit (MMU), a hardware component present in modern CPUs. The MMU performs address translation based on mappings maintained by the OS.

3. **Demand Paging**: Virtual memory systems often employ demand paging to optimize memory usage. With demand paging, not all pages of a process are loaded into physical memory at once. Instead, pages are loaded on-demand, as they are accessed by the process.

This allows for more efficient usage of physical memory by prioritizing the loading of frequently accessed pages.

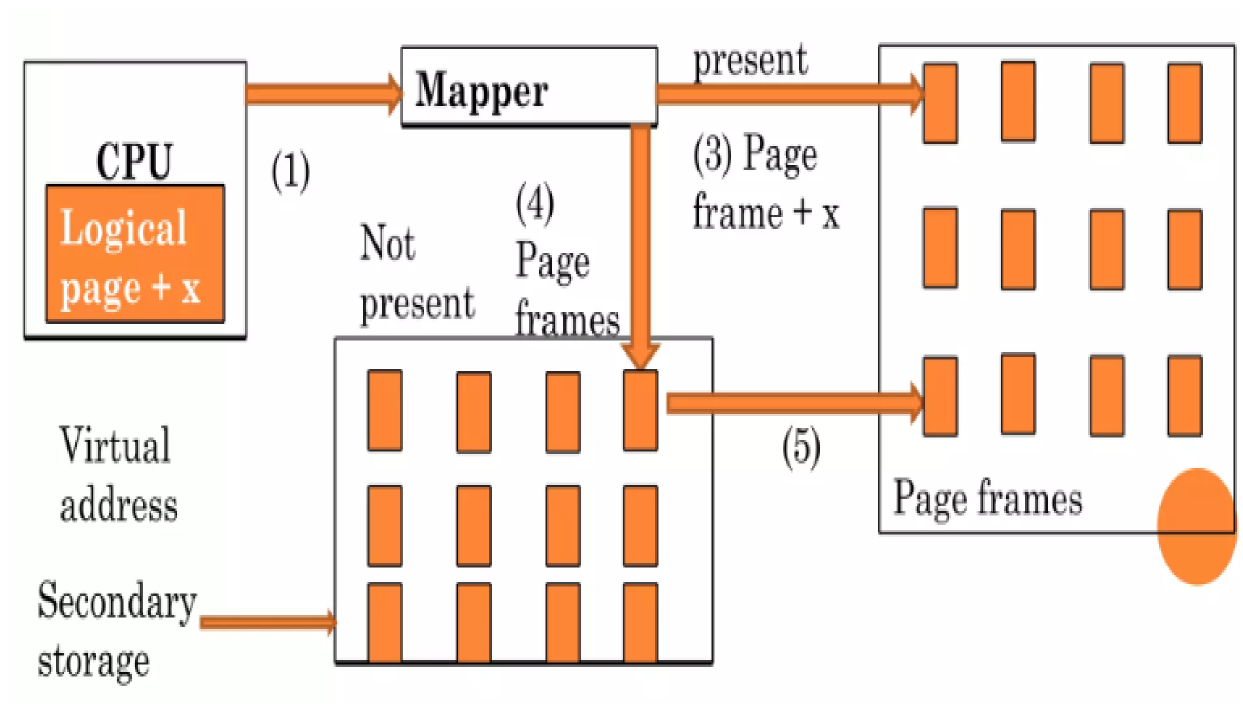
4. **Page Replacement Algorithms**: When physical memory becomes scarce, the OS must decide which pages to evict from memory to make room for incoming pages. Page replacement algorithms, such as Least Recently Used (LRU), First-In-First-Out (FIFO), or Clock algorithms, are used to select pages for eviction based on various criteria such as access frequency or recency.

5. **Swap Space Management**: In addition to physical memory, the OS maintains a swap space on disk to store pages that are not currently in physical memory. When a page is evicted from memory, it is written to the swap space. Conversely, pages are loaded from the swap space into physical memory when needed. Effective swap space management is critical for optimizing system performance and preventing excessive disk I/O.

6. **Memory Protection**: Virtual memory systems provide memory protection mechanisms to prevent unauthorized access to memory regions. Each process operates within its own virtual address space, isolated from other processes. Memory protection features, such as read-only and no-execute permissions, ensure that processes cannot inadvertently modify or execute unauthorized memory regions.

7. **Shared Memory**: Virtual memory systems support shared memory mechanisms, allowing multiple processes to access the same physical memory region concurrently. This facilitates inter-process communication and collaboration, enabling efficient data sharing between processes without the need for explicit data copying.

System Architecture



The system architecture of a virtual memory management system in an operating system involves several components working together to provide efficient memory abstraction, allocation, and management. Here's an overview of the typical architecture:

1. **Memory Management Unit (MMU)**:
    - The MMU is a hardware component integrated into the CPU.
    - It translates virtual addresses generated by the CPU into physical addresses.
    - MMU operates based on page tables maintained by the OS.

2. **Page Tables**:
    - Page tables are data structures maintained by the operating system to map virtual addresses to physical addresses.
    - Each process has its own page table, which defines the mapping between its virtual addresses and physical addresses.
    - Page tables are stored in memory and accessed by the MMU during address translation.

3. **Page Fault Handler**:
    - When a process accesses a memory page that is not currently in physical memory, a page fault occurs.
    - The page fault handler is a component of the OS responsible for handling page faults.
    - It retrieves the required page from disk (if necessary) and updates the page table to reflect the new mapping.

4. **Demand Paging Mechanism**:
    - Demand paging is a technique used to load pages into physical memory only when they are needed.
    - When a page is accessed by a process and is not present in physical memory, the demand paging mechanism triggers the page fault handler to bring the page into memory.

5. **Page Replacement Algorithm**:
    - Page replacement algorithms are used to select pages for eviction when physical memory becomes full.
    - Common algorithms include Least Recently Used (LRU), First-In-First-Out (FIFO), and Clock algorithm.
    - The selected page is written to disk if it has been modified (dirty), and the corresponding entry in the page table is updated.

6. **Swap Space**:
    - Swap space is an area on disk used to store pages that are not currently in physical memory.
    - When a page is evicted from memory, it is written to swap space.
    - Pages are fetched from swap space into physical memory when needed.

7. **Memory Protection Mechanisms**:

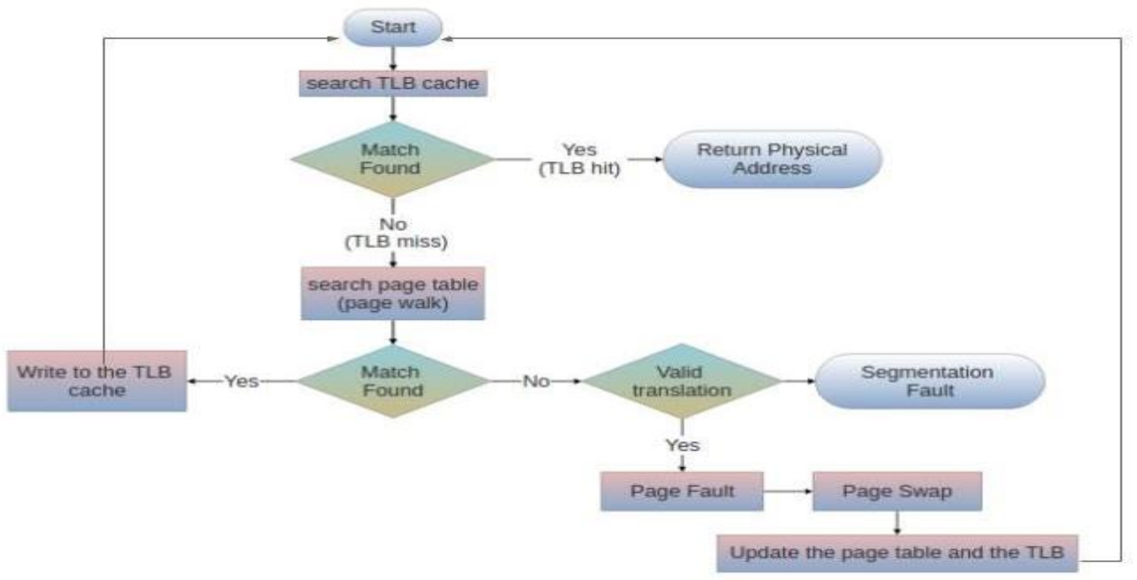
- Memory protection mechanisms ensure that processes cannot access unauthorized memory regions.
- Permissions such as read-only, read-write, and execute permissions are enforced by the MMU based on settings in the page table entries.

8. **Shared Memory Management**:

   - Shared memory mechanisms allow multiple processes to share a common memory region.
   - The OS ensures proper synchronization and access control to prevent conflicts between processes accessing shared memory.

Methodology

The methodology used in virtual memory management systems involves a combination of algorithms, techniques, and data structures to efficiently manage the virtual-to-physical memory mapping and handle memory operations. Implementing a virtual memory management system involves a systematic approach to efficiently manage the mapping between virtual and physical memory, as well as handling memory operations in an operating system. The methodology typically begins with designing data structures such as page tables to represent the mapping between virtual and physical memory addresses. Algorithms for page replacement, such as LRU or FIFO, are selected to optimize memory usage and minimize page faults. Demand paging strategies are employed to load pages into memory only when needed, conserving resources and improving performance. Hardware support like TLB caches is utilized to accelerate address translation, while memory protection mechanisms are implemented to ensure process isolation and security. Swapping techniques may be employed to move inactive processes or pages to disk storage to free up physical memory. Additionally, optimizations such as copy-on-write and memory-mapped files may be integrated to further enhance memory utilization and I/O performance.



Systems calls used

1. File I/O:

- The program reads input from a file using the `open()` function to open the file for reading (`'r'` mode) and then uses the `readlines()` method to read lines from the file.
- Similarly, the program might use the `open()` function to write output to a file.

2. Threading:

- The program utilizes threading to create concurrent execution flows. Threads are created using the `threading.Thread()` constructor and started using the `start()` method.
- Threading is used to simulate the concurrent execution of the memory management unit (MMU) and the OS scheduler.

3. **Time-related functions:**

- Time-related functions like `time.sleep()` are used to introduce delays or wait periods in the program execution. For example, in the `scheduler()` function, `time.sleep(5)` is used to introduce a 5-second delay between scheduling iterations.

4. **OS exit:**

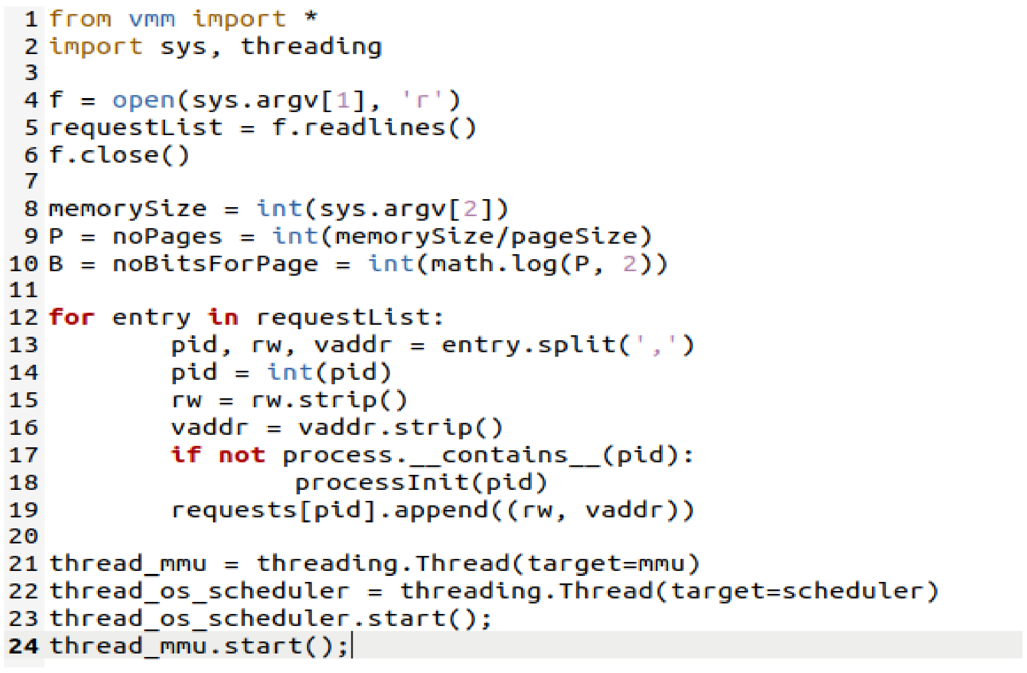
- The `os._exit(0)` call is used to exit the program and terminate the execution of the operating system.

5. Signal Handling:

- The program defines a custom `SIGUSR1` signal using the `SignalUser1()` class. While not a traditional OS system call, signal handling is a mechanism provided by the operating system to notify processes of asynchronous events.
- The program simulates a signal handler function (`SIGUSR1_Handler`) that gets executed when a specific event occurs, such as handling page faults.

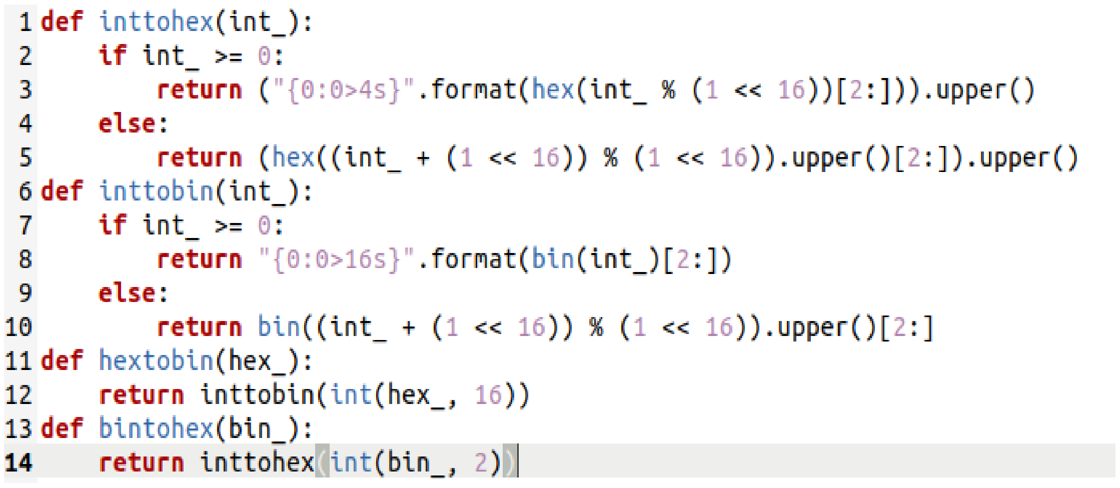
Source code

main.py



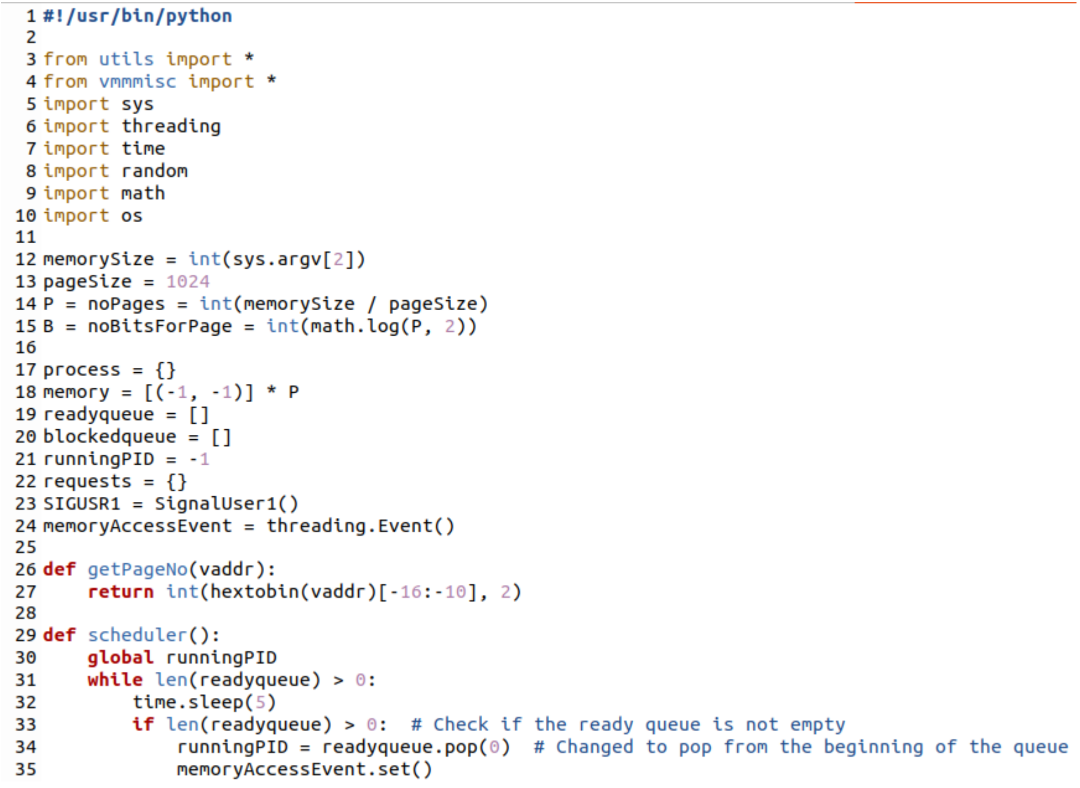
```python
from vmm import *
import sys, threading

f = open(sys.argv[1], 'r')
requestList = f.readlines()
f.close()

memorySize = int(sys.argv[2])
P = noPages = int(memorySize/pageSize)
B = noBitsForPage = int(math.log(P, 2))

for entry in requestList:
        pid, rw, vaddr = entry.split(',')
        pid = int(pid)
        rw = rw.strip()
        vaddr = vaddr.strip()
        if not process.__contains__(pid):
                processInit(pid)
        requests[pid].append((rw, vaddr))

thread_mmu = threading.Thread(target=mmu)
thread_os_scheduler = threading.Thread(target=scheduler)
thread_os_scheduler.start();
thread_mmu.start();
```

utils.py

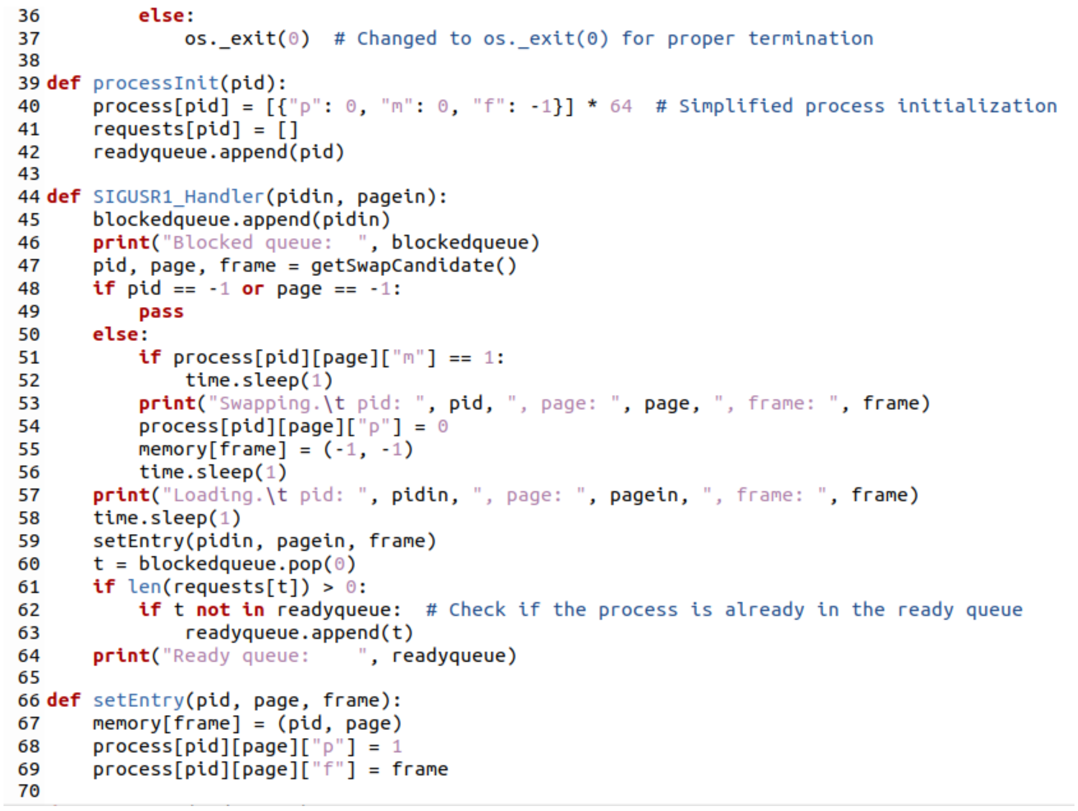


```python
def inttohex(int_):
    if int_ >= 0:
        return ("{0:0>4s}".format(hex(int_ % (1 << 16))[2:])).upper()
    else:
        return (hex((int_ + (1 << 16)) % (1 << 16)).upper()[2:]).upper()
def inttobin(int_):
    if int_ >= 0:
        return "{0:0>16s}".format(bin(int_)[2:])
    else:
        return bin((int_ + (1 << 16)) % (1 << 16)).upper()[2:]
def hextobin(hex_):
    return inttobin(int(hex_, 16))
def bintohex(bin_):
    return inttohex(int(bin_, 2))
```

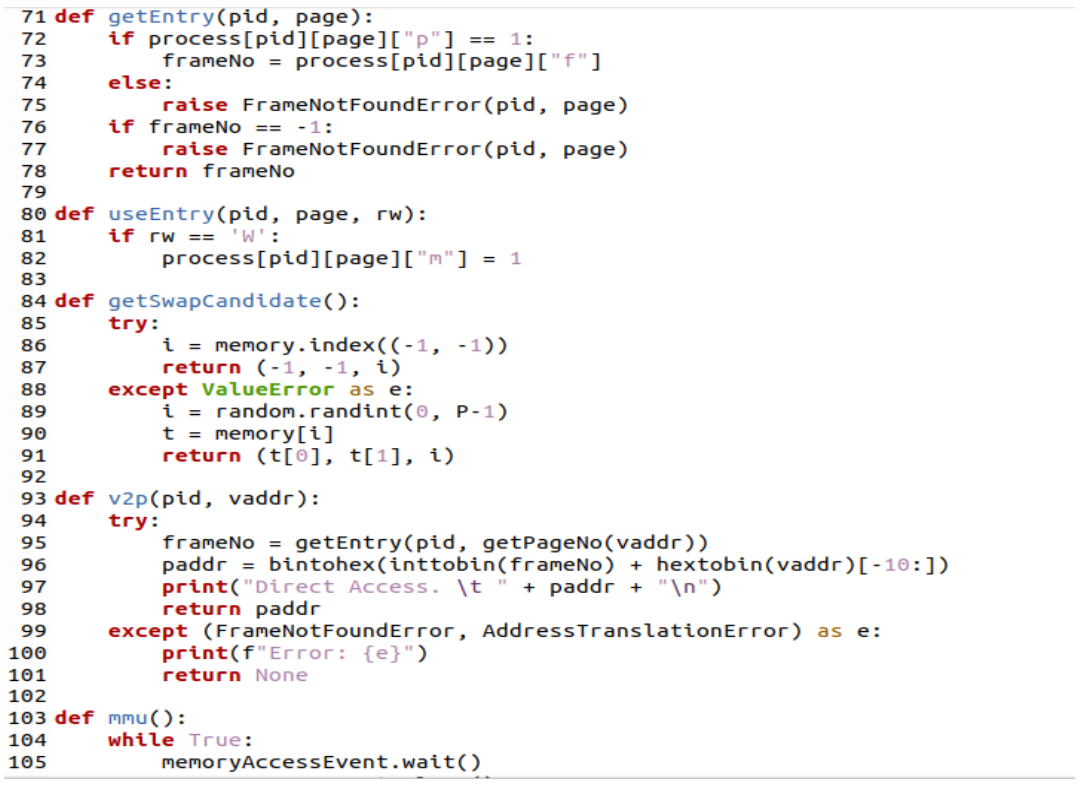
vmm.py



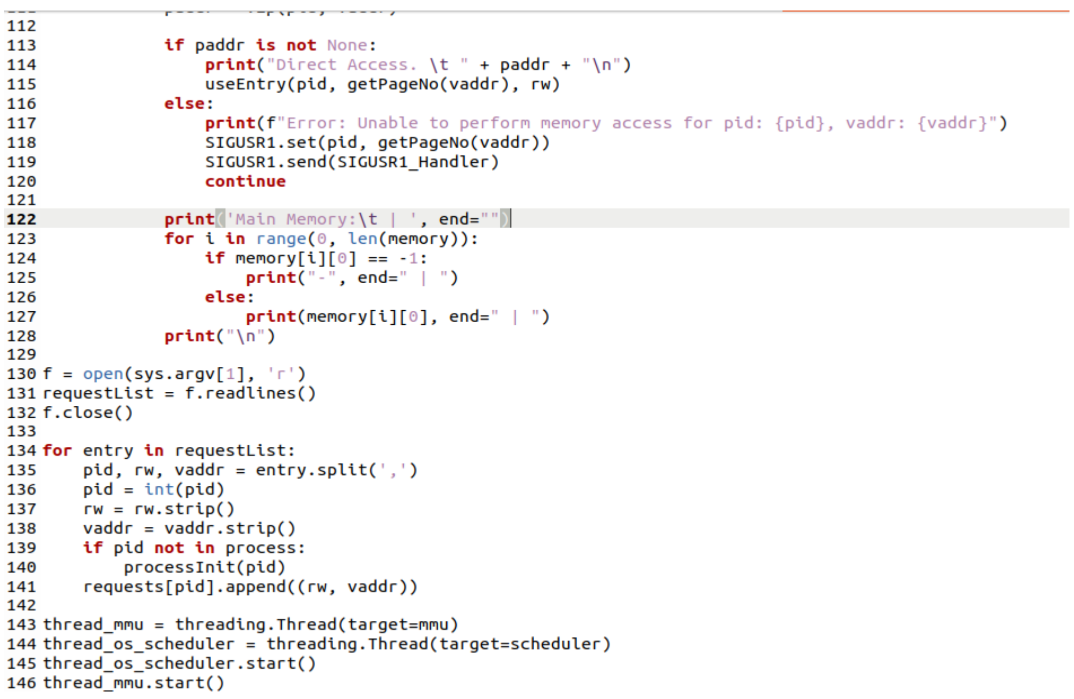
```python
#!/usr/bin/python

from utils import *
from vmmisc import *
import sys
import threading
import time
import random
import math
import os

memorySize = int(sys.argv[2])
pageSize = 1024
P = noPages = int(memorySize / pageSize)
B = noBitsForPage = int(math.log(P, 2))

process = {}
memory = [(-1, -1)] * P
readyqueue = []
blockedqueue = []
runningPID = -1
requests = {}
SIGUSR1 = SignalUser1()
memoryAccessEvent = threading.Event()

def getPageNo(vaddr):
    return int(hextobin(vaddr)[-16:-10], 2)

def scheduler():
    global runningPID
    while len(readyqueue) > 0:
        time.sleep(5)
        if len(readyqueue) > 0:  # Check if the ready queue is not empty
            runningPID = readyqueue.pop(0)  # Changed to pop from the beginning of the queue
            memoryAccessEvent.set()
```



```python
        else:
            os._exit(0)  # Changed to os._exit(0) for proper termination

def processInit(pid):
    process[pid] = [{"p": 0, "m": 0, "f": -1}] * 64  # Simplified process initialization
    requests[pid] = []
    readyqueue.append(pid)

def SIGUSR1_Handler(pidin, pagein):
    blockedqueue.append(pidin)
    print("Blocked queue:  ", blockedqueue)
    pid, page, frame = getSwapCandidate()
    if pid == -1 or page == -1:
        pass
    else:
        if process[pid][page]["m"] == 1:
            time.sleep(1)
        print("Swapping.\t pid: ", pid, ", page: ", page, ", frame: ", frame)
        process[pid][page]["p"] = 0
        memory[frame] = (-1, -1)
        time.sleep(1)
    print("Loading.\t pid: ", pidin, ", page: ", pagein, ", frame: ", frame)
    time.sleep(1)
    setEntry(pidin, pagein, frame)
    t = blockedqueue.pop(0)
    if len(requests[t]) > 0:
        if t not in readyqueue:  # Check if the process is already in the ready queue
            readyqueue.append(t)
    print("Ready queue:     ", readyqueue)

def setEntry(pid, page, frame):
    memory[frame] = (pid, page)
    process[pid][page]["p"] = 1
    process[pid][page]["f"] = frame

```

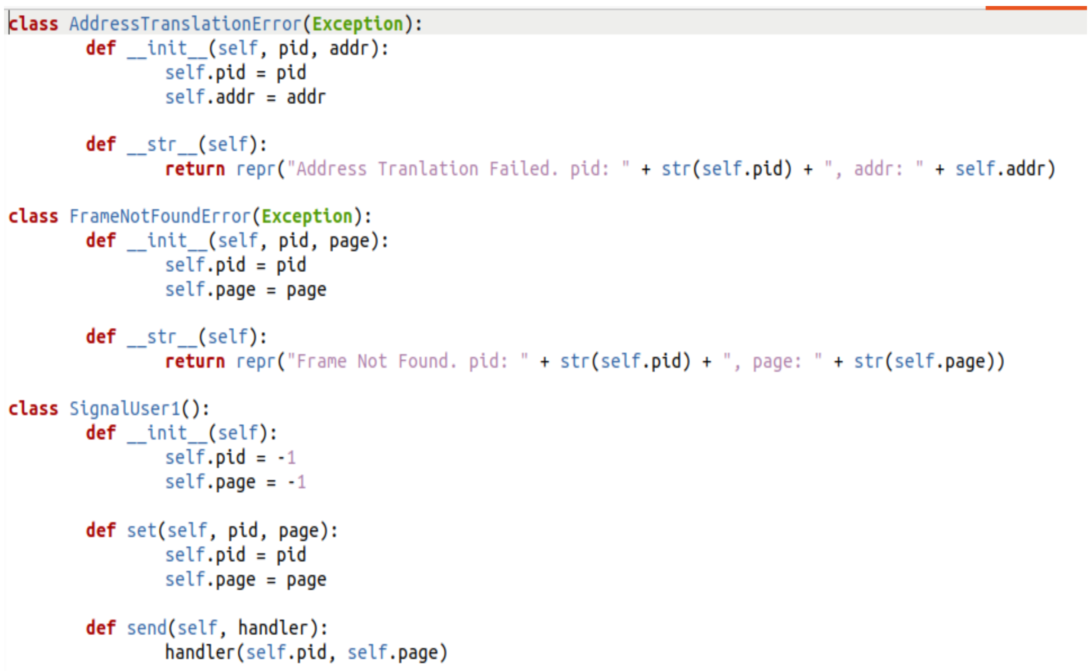


```python
def getEntry(pid, page):
    if process[pid][page]["p"] == 1:
        frameNo = process[pid][page]["f"]
    else:
        raise FrameNotFoundError(pid, page)
    if frameNo == -1:
        raise FrameNotFoundError(pid, page)
    return frameNo

def useEntry(pid, page, rw):
    if rw == 'W':
        process[pid][page]["m"] = 1

def getSwapCandidate():
    try:
        i = memory.index((-1, -1))
        return (-1, -1, i)
    except ValueError as e:
        i = random.randint(0, P-1)
        t = memory[i]
        return (t[0], t[1], i)

def v2p(pid, vaddr):
    try:
        frameNo = getEntry(pid, getPageNo(vaddr))
        paddr = bintohex(inttobin(frameNo) + hextobin(vaddr)[-10:])
        print("Direct Access. \t " + paddr + "\n")
        return paddr
    except (FrameNotFoundError, AddressTranslationError) as e:
        print(f"Error: {e}")
        return None

def mmu():
    while True:
        memoryAccessEvent.wait()
```



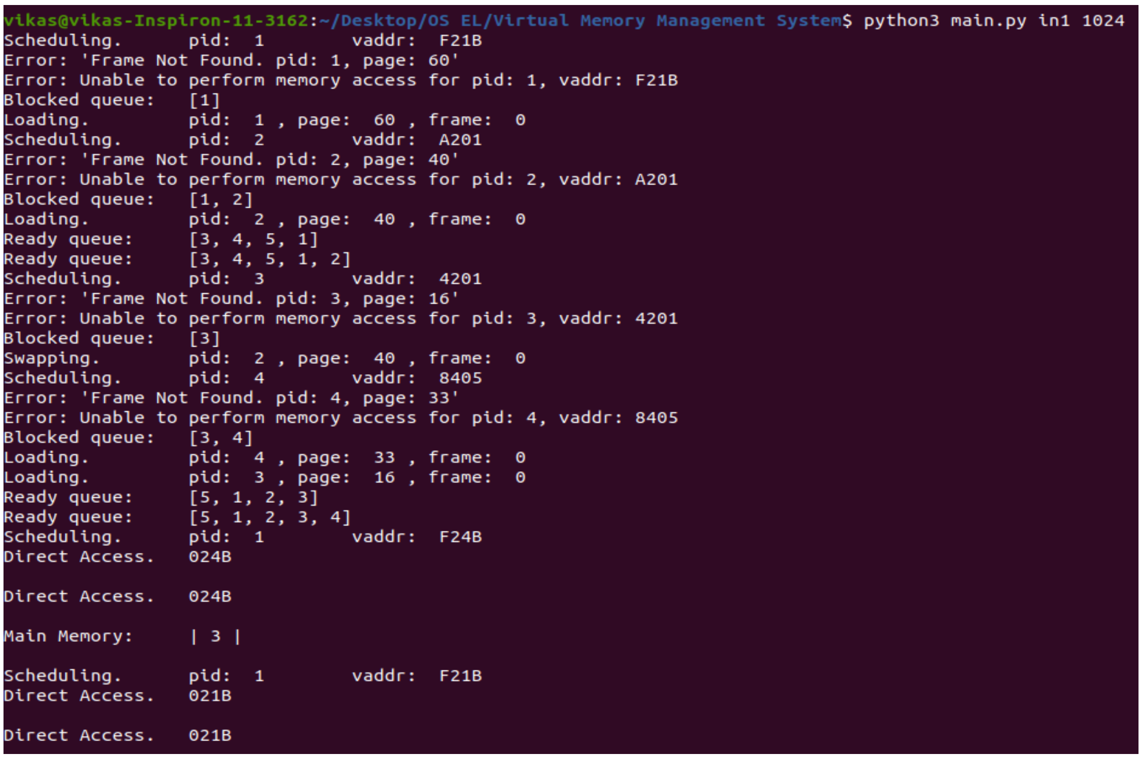
```
            if paddr is not None:
                print("Direct Access. \t " + paddr + "\n")
                useEntry(pid, getPageNo(vaddr), rw)
            else:
                print(f"Error: Unable to perform memory access for pid: {pid}, vaddr: {vaddr}")
                SIGUSR1.set(pid, getPageNo(vaddr))
                SIGUSR1.send(SIGUSR1_Handler)
                continue

            print('Main Memory:\t | ', end="")
            for i in range(0, len(memory)):
                if memory[i][0] == -1:
                    print("-", end=" | ")
                else:
                    print(memory[i][0], end=" | ")
            print("\n")

f = open(sys.argv[1], 'r')
requestList = f.readlines()
f.close()

for entry in requestList:
    pid, rw, vaddr = entry.split(',')
    pid = int(pid)
    rw = rw.strip()
    vaddr = vaddr.strip()
    if pid not in process:
        processInit(pid)
    requests[pid].append((rw, vaddr))

thread_mmu = threading.Thread(target=mmu)
thread_os_scheduler = threading.Thread(target=scheduler)
thread_os_scheduler.start()
thread_mmu.start()
```

vmmmisc.py

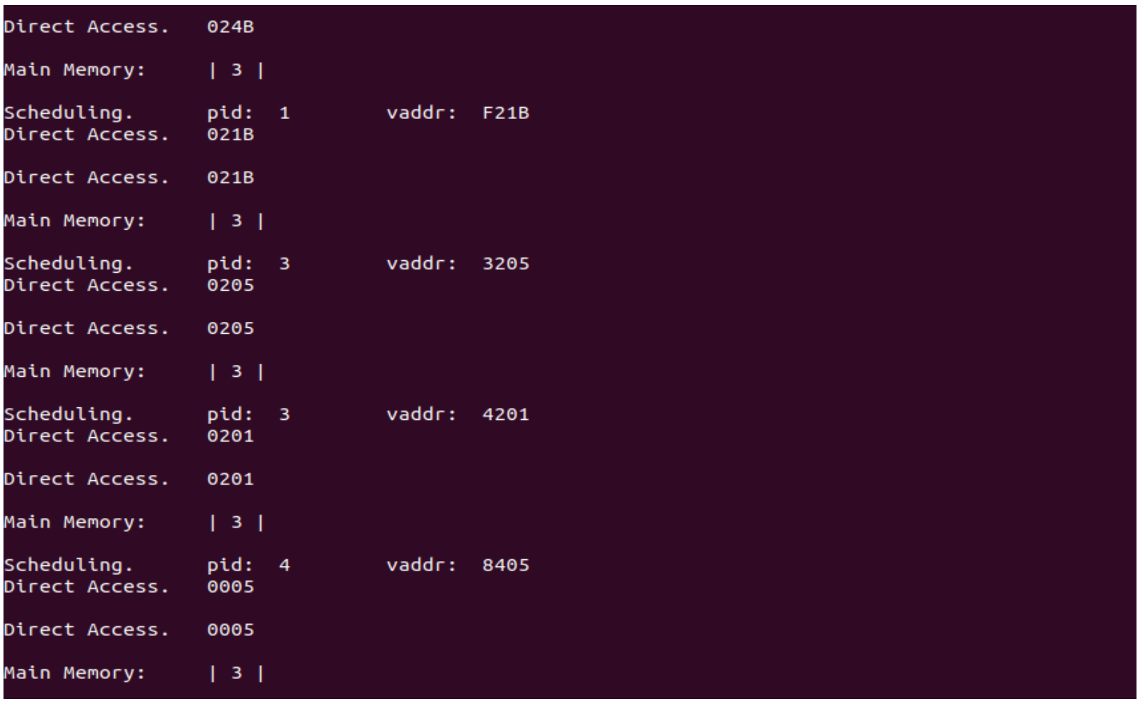


```
class AddressTranslationError(Exception):
        def __init__(self, pid, addr):
                self.pid = pid
                self.addr = addr

        def __str__(self):
                return repr("Address Tranlation Failed. pid: " + str(self.pid) + ", addr: " + self.addr)

class FrameNotFoundError(Exception):
        def __init__(self, pid, page):
                self.pid = pid
                self.page = page

        def __str__(self):
                return repr("Frame Not Found. pid: " + str(self.pid) + ", page: " + str(self.page))

class SignalUser1():
        def __init__(self):
                self.pid = -1
                self.page = -1

        def set(self, pid, page):
                self.pid = pid
                self.page = page

        def send(self, handler):
                handler(self.pid, self.page)
```

Output/results:



```
vikas@vikas-Inspiron-11-3162:~/Desktop/OS EL/Virtual Memory Management System$ python3 main.py in1 1024
Scheduling.      pid:  1        vaddr:  F21B
Error: 'Frame Not Found. pid: 1, page: 60'
Error: Unable to perform memory access for pid: 1, vaddr: F21B
Blocked queue:   [1]
Loading.         pid:  1 , page:  60 , frame:  0
Scheduling.      pid:  2        vaddr:  A201
Error: 'Frame Not Found. pid: 2, page: 40'
Error: Unable to perform memory access for pid: 2, vaddr: A201
Blocked queue:   [1, 2]
Loading.         pid:  2 , page:  40 , frame:  0
Ready queue:     [3, 4, 5, 1]
Ready queue:     [3, 4, 5, 1, 2]
Scheduling.      pid:  3        vaddr:  4201
Error: 'Frame Not Found. pid: 3, page: 16'
Error: Unable to perform memory access for pid: 3, vaddr: 4201
Blocked queue:   [3]
Swapping.        pid:  2 , page:  40 , frame:  0
Scheduling.      pid:  4        vaddr:  8405
Error: 'Frame Not Found. pid: 4, page: 33'
Error: Unable to perform memory access for pid: 4, vaddr: 8405
Blocked queue:   [3, 4]
Loading.         pid:  4 , page:  33 , frame:  0
Loading.         pid:  3 , page:  16 , frame:  0
Ready queue:     [5, 1, 2, 3]
Ready queue:     [5, 1, 2, 3, 4]
Scheduling.      pid:  1        vaddr:  F24B
Direct Access.   024B

Direct Access.   024B

Main Memory:     | 3 |

Scheduling.      pid:  1        vaddr:  F21B
Direct Access.   021B

Direct Access.   021B
```



```
Direct Access.   024B

Main Memory:     | 3 |

Scheduling.      pid:  1        vaddr:  F21B
Direct Access.   021B

Direct Access.   021B

Main Memory:     | 3 |

Scheduling.      pid:  3        vaddr:  3205
Direct Access.   0205

Direct Access.   0205

Main Memory:     | 3 |

Scheduling.      pid:  3        vaddr:  4201
Direct Access.   0201

Direct Access.   0201

Main Memory:     | 3 |

Scheduling.      pid:  4        vaddr:  8405
Direct Access.   0005

Direct Access.   0005

Main Memory:     | 3 |
```

Processes: Represented by PIDs (Process IDs) like 1, 2, 3, 4, and 5. These processes need to access memory for their tasks.

Memory: Divided into pages. The output shows a small main memory with a capacity of one page at a time.

Frames: Represent slots in main memory where pages can be loaded.

Scheduling: The system selects a process to run.

vaddr: This refers to the virtual address the process wants to access.

Error: 'Frame Not Found': The requested page is not currently in main memory.

Loading: The system loads the requested page from secondary storage (like a disk) into a free frame.

Swapping: To free up a frame, the system swaps a page from main memory back to secondary storage.

Blocked queue: Processes waiting for a page to be loaded are added to this queue.

Ready queue: Processes that have all their required pages in memory and are ready to run are added to this queue.

Direct Access: The process successfully accesses the data in memory.

Long Access: The process performs a longer access on the data.

Process 1: Attempts to access virtual address `F21B`, but encounters a "Frame Not Found" error. It gets blocked.

Process 2: Tries to access virtual address `A201`, also encountering a "Frame Not Found" error and gets blocked.

Process 3: Seeks to access virtual address `4201`, facing the same "Frame Not Found" error and getting blocked.

Process 2 (Swapping): Since it was blocked, it's swapped out to make space for other processes.

Process 4: Like the previous processes, it tries to access virtual address `8405`, encountering the "Frame Not Found" error and getting blocked.

Process 4 Loading: After being blocked, Process 4 is loaded into memory, along with Process 3.

Ready Queue: Processes 1, 2, and 3 are ready for execution, along with Processes 4 and 5.

Process 1 Access: Finally, Process 1 accesses virtual address `F24B`, followed by `F21B`.

Process 3 Access: It accesses virtual addresses `3205` and `4201`.

Process 4 Access: Process 4 accesses virtual address `8405`.

Main Memory: Displays the content of the main memory after each access.

Conclusion

In conclusion, the provided log offers insights into the virtual memory management and scheduling mechanisms within an operating system environment. It demonstrates the orchestration of processes, memory access handling, and the dynamic allocation of memory resources. Despite encountering "Frame Not Found" errors, processes are appropriately managed through blocking, loading, and swapping mechanisms, ensuring efficient utilization of available memory. Moreover, the direct access operations showcase successful memory accesses once processes are loaded into memory. This implementation underscores the importance of robust memory management strategies, including paging, swapping, and process scheduling algorithms, in ensuring the smooth execution of tasks within a multitasking computing environment.

References

1. Lee and D. Wang, "**Enhancing Virtual Memory Efficiency-Through Machine Learning Techniques**," in Proc. USENIX Annual Technical Conference, July 2023.

2. Dixit, Shridhar S., **“Study of Virtual Memory”** (2020). Retrospective Theses and Dissertations. 619. https://stars.library.ucf.edu/rtd/619

3. Patel and B. Kumar, "**Optimizing Virtual Memory Management for Big Data Applications**," in Proc. ACM Symposium on Operating Systems Principles (SOSP), October 2022.